History-Aware, Resource-Based Dynamic Scheduling For Heterogeneous Multi-core Processors

Ali Z. Jooya
Computer Engineering Department
Iran University of Science and Technology
School of Computer Science of IPM
Tehran, Iran
al_zolfaghari@comp.iust.ac.ir

Amirali Baniasadi[+]
Electrical and Computer Engineering Department
University of Victoria
Victoria, British Columbia, Canada
amirali@ece.uvic.ca

Morteza Analoui
Computer Engineering Department
Iran University of Science and Technology
Tehran, Iran
analoui@iust.ac.ir

Abstract

In this work we introduce a history-aware, resource-based dynamic (or simply HARD) scheduler for heterogeneous CMPs. HARD relies on recording application resource utilization and throughput to adaptively change cores for applications during runtime. We show that HARD can be configured to achieve both performance and power improvements. We compare HARD to a complexity-based static scheduler and show that HARD outperforms this alternative.

1. Introduction

Exploiting thread-level parallelism is believed to be a reliable way to achieve higher performance improvements in the future. Moreover, as technology advances provide microprocessor design with more options, finding new solutions to use the possible capabilities is necessary. Chip multiprocessing offers an attractive solution as using multiple cores makes efficient execution of parallel threads possible.

There are many design decisions that impact performance in a CMP. Designers have to make many choices including, the number of cores, the appropriate interconnect and the memory system. Server applications focus primarily on throughput. A CMP targeting these applications, ideally, uses a large number of small low-power cores. On the other hand, for desktop users the performance of a single application is more important. Architects of a desktop would more likely focus on a smaller number of larger and more complex cores with better single-thread performance. As CMPs become more popular in different design spaces, and considering the variety of programs that a typical processor is expected to run, exploiting heterogeneous CMPs seems to be a reasonable future choice. A heterogeneous CMP equipped with both high- and low-complexity cores provides enough resources to execute both latency- and throughput-sensitive applications efficiently. Recent research in heterogeneous CMPs has identified significant advantages over homogeneous CMPs in terms of power and throughput [1].

In order to achieve the potential benefits of heterogonous CMPs, an effective application-to-core scheduler is essential. Such a scheduler would assign latency-sensitive applications to stronger cores while leaving the throughput-sensitive to the weaker ones. Running demanding applications on weak cores hurts performance. Meantime, running low-demand applications on strong cores, results in unnecessary power dissipation. The task of the scheduler is to avoid both scenarios, finding the best assignment that prevents both resource over-utilization and under-utilization.

Between two possible scheduling policies, static and dynamic, the latter has significant advantages. This is particularly true for heterogeneous CMPs. In addition to the behavior variations among applications, there are behavior changes within an application. Therefore, an application's demand for processor resources varies during runtime. Dynamic scheduling uses runtime information to tune core-application assignments through such changes.

Many scheduling policies have been introduced for parallel programs, (e.g., gang scheduling and uncoordinated scheduling). Such policies often focus on concurrent execution on the threads of an application on distinct processors. In this study applications are independent.

[+] The author was spending his sabbatical leave at the school of computer science of IPM when this work was done.

We introduce a **h**istory-**a**ware, **r**esource-based **d**ynamic (or simply *HARD*) scheduler for heterogeneous CMPs. We record and use past core assignments for an application to find the best matching core. Depending on how we set our system parameters we can aim at improving performance or reducing power. We improve overall performance by *upgrading* applications requiring more resources to stronger cores. We reduce power dissipation by *downgrading* under-utilizing applications to weaker cores.

We evaluate our scheduler from two perspectives: user's and system's. We use Average Normalized Turnaround Time (*ANTT*) for the user's perspective and System Throughput (*STP*) for the system's perspective [2]. Note that the *Fairness* of the scheduler can affect the overall throughput and power consumption. Therefore, and to provide better understanding, we also report *Fairness* for our scheduler.

The rest of the paper has the following organization. Section 2 describes the motivation of our study. In section 3 we introduce HARD in more detail. In section 4 we present methodology and results. Section 5 summarizes the results and contributions of our work. In section 6 we offer concluding remarks.

2. Motivation

Application behavior changes during run time resulting in variations in application demand for processor resources during different phases. Some phases may weakly utilize processor resources while other phases may over-utilize them.

Figures 1 and 2 show behaviour changes in terms of processor throughput and core utilization, respectively. We have divided the application execution time into 100K clock cycle intervals. We use the number of retired instructions during an interval to estimate application throughput. We define core utilization as the ratio of the average instruction windows occupancy to the instruction window size. Figure 1 illustrates the throughput for fmm and barnes applications. Figure 2 shows integer and floating point core utilizations for the same applications. We conclude from Figures 1 and 2 that applications show periodic and variable behaviour during runtime. This is consistent with previous studies [3-5].

Application behavior and resource demand variations have motivated designing CMPs where different cores come with different complexities and resources. Heterogeneous CMPs run different applications on different cores with the goal to maximize system throughput while improving resource utilization. This goal could be achieved if more demanding applications/threads are assigned to more complex and powerful cores while less demanding applications/threads are assigned to smaller and simpler cores.

To achieve best results, applications have to be assigned to cores with different complexities as application demand changes. Under such circumstances, a dynamic scheduler can reassign applications to cores according to resource requirements. Previous study has shown that heterogeneous CMPs equipped with dynamic scheduling increase maximum system throughput, improve average response time and remain stable under heavy loads [6].

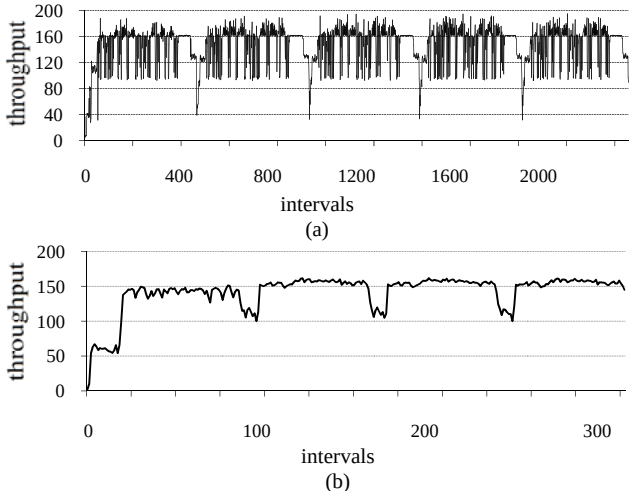

Figure 1. Throughput for a) fmm and b) barnes applications. The x-axis shows the execution intervals and the y-axis shows the system throughput divided by thousand.

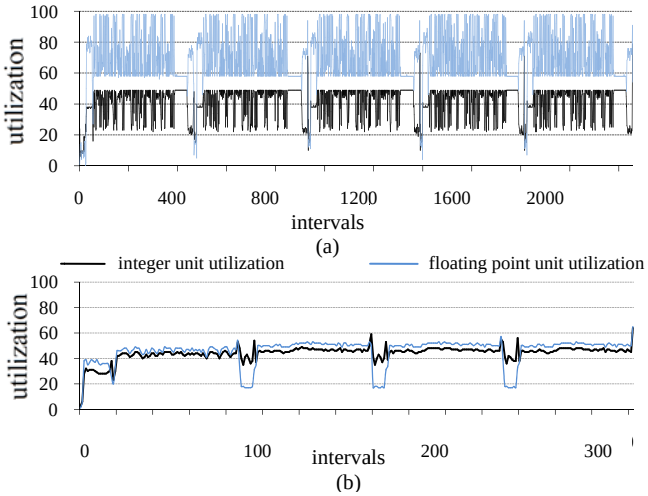

Figure 2. Core utilization for a) fmm and b) barnes applications for integer and floating point instruction buffers. The x-axis shows the execution intervals.

HARD comes with two important benefits. The first benefit is maximizing throughput. This is achieved as more demanding threads are assigned to more powerful cores as soon as such demands are noted and suitable cores become available. The second benefit is reducing power. The scheduler assigns threads and applications to more simple cores as soon as the system detects that the application is underutilizing its core. In this work we show that both benefits are obtainable using HARD.

3. HARD Scheduler

In this section we introduce HARD. The scheduler relies on two subsections: phase detection and reassignment.

3.1. Phase Detection Unit

In order to record application behavior we run our phase detection algorithm at the end of every 100K clock cycle interval. Very short intervals can degrade performance as the result of frequent and high switching overhead. Long intervals, on the other hand, may miss application phase changes. We picked 100K intervals after testing many alternatives.

The phase detection algorithm relies on two measurements: throughput and core utilization. In [17] Kumar used throughput to detect phase changes. We use throughput and core resource utilization to identify a phase change. We use a counter to record the number of retired instructions as an estimation of application throughput. The size of the counter depends on the maximum possible throughput in each core. For example, maximum nominal throughput of a 6-way core during a 100K clock interval is 600K instructions. Therefore, a 20-bit binary counter is large enough to store the throughput of the intervals.

To calculate core utilization, instruction window occupancy is measured. Two counters, one for integer instruction window utilization and one for floating point instruction window utilization, are used to estimate core utilization. The phase detection algorithm uses the greater of the two to decide core utilization. The sizes of the counters depend on both instruction windows size and interval size. For example, for a core with the largest instruction window among all cores (i.e., 104 and 48 for integer and floating point instructions respectively), and 100K clock intervals, the maximum number of occupied instruction window entries are 104×100K and 48×100k, respectively. These numbers can be stored using 24-bit and 23-bit binary counters. Core utilization within each interval is estimated by dividing the counter values by the interval size.

The phase detection algorithm uses throughput and core utilization to categorize intervals into one of the following classes.

- *Upgrading* (UG): An interval belongs to this category if a) the utilization exceeds high utilization threshold (HUT) or b) the throughput exceeds the high throughput threshold (HTT). Either condition implies that the application could use more resources and that switching to a stronger core will most likely boost performance.
- *Downgrading* (DG): An interval belongs to this category if utilization is less than the low utilization threshold (LUT) and throughput is less than the low throughput threshold (LTT). Under these circumstances we assume that the application is under-utilizing core resources and switching it to a weaker core will most likely reduce power dissipation without compromising performance.
- *No-change* (NC): An interval not belonging to either of the classes discussed above is assumed to be in this class. We assume that the core is running the application within reasonable power and performance budgets. Therefore there is no need to switch to a weaker or stronger core.

Figure 3 shows these three classes.

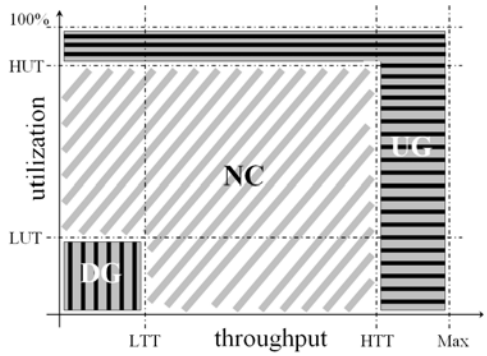

Figure 3. Three intervals classes. The x-axis represents throughput and the y-axis represents core utilization.

3.2. Reassignment Unit

As we discussed before, our dynamic scheduler uses the phase detection algorithm to detect application phase changes. When the phase detection unit detects an upgrading or downgrading phase change, the scheduler activates the reassignment unit. This unit records application phase change history. History is used to decide if the application needs to switch to another core.

Note that there is a cost associated with switching. For example, the cache should be flushed to save all dirty

cache data in the shared L2. The cold-start effect on caches is another unwanted consequence [17]. The simulator we utilize applies these overheads.

The reassignment unit relies on the phase detection unit output and the value of a 5-bit counter referred to as the demand history counter (DHC). Each core has its own demand counter which is used to keep record of the phase change history for each application. DHC is incremented or decremented if the last interval was classified as either UG or DG respectively. The counter value is used to decide if switching to a stronger or weaker core is necessary. We switch the application to a stronger or weaker core if this counter's value is above or below a pre-decided threshold respectively (we picked these thresholds to 6 and -6 after testing many alternatives).

We run the phase detection algorithm every 100K. However, switching to a new core requires up to six consecutive intervals until DHC reaches one of the pre-decided thresholds. It is important to notice that using DHC masks the transitive change in applications behavior effectively minimizing oscillation frequency.

Our study shows that NC intervals require special attention to make sure that the scheduler is effective. In our design we reset the demand counter if and when a number of consecutive NC intervals occur. Our study shows that best results are obtained if the counter is reset after four consecutive NC intervals.

As the demand counter reaches 6 (or -6), the scheduler should pick a stronger (or weaker) core. An application can move only one level at a time. For instance, consider a CMP with cores representing three levels of performance, and a situation where a UG application is running on the weakest core. The reassignment unit can only assign the core to one of the mid-level cores. There may be more than one choice at this stage. In the event where one or more idle core exists, the scheduler transfers the application from the original core to one of the idle cores (in our study the core with the lowest ID is picked). If all the possible target cores are busy, the scheduler has to choose one of the busy cores and swap the two applications. The scheduler picks the target core as follows. When upgrading to a stronger core, the (busy) core with the least demand counter is picked. When downgrading to a weaker core, the (busy) core with the greatest demand counter is picked. Intuitively we do so to make sure applications running on the two swapping cores benefit most (or harm least in some cases as explained below).

In some cases swapping the two applications may degrade the overall throughput of the system. For example, consider an upgrading switch where the demand counter on the target core is equal or greater than six. The application on the target core is also an upgrading one and will most likely lose performance if the switch takes place. To avoid such circumstances, the scheduler avoids the switch if the target core has to make a move in the opposite direction of that suggested by its demand counter. For each interval that the switch request of the application does not take place, the scheduler increments DHC until the counter reaches a maximum (10 in this study). The following UG intervals can not change DHC after reaching the maximum. At this point the scheduler is aware that the application needs an upgrading switch as soon as possible. The minimum value of DHC is -10.

To provide better insight in Figure 4 we report DHC values for one strong core and one weak core. This data is extracted from running five benchmarks (mix3, see section 4 for more details) on a five-core CMP. As each benchmark may run on different cores during its execution time, the intervals belong to the execution of different benchmarks coming and leaving the core. Figure 4(a) shows DHC values for the strongest core over 850 intervals. The core time is divided between water-spatial (90% of intervals) and cholesky (10%). A DHC value of 10 show upgrading switch requests (mostly from water-spatial), which are impossible on this processor. This condition results in exclusive usage of a core by one application as the scheduler cannot switch any application to a core with DHC equal to 10. Therefore, DHC remains 10 until the application changes its phase decreasing DHC with every DG interval and providing a chance for a switch to other cores. Figure 4(b) shows a similar trend for the weakest core (intervals with DHC equal to -10) over 400 intervals. The intervals are divided among all benchmarks, except ocean. Most of the intervals with DHC of -10 belong to cholesky. The rest (including the beginning intervals) belong to barnes.

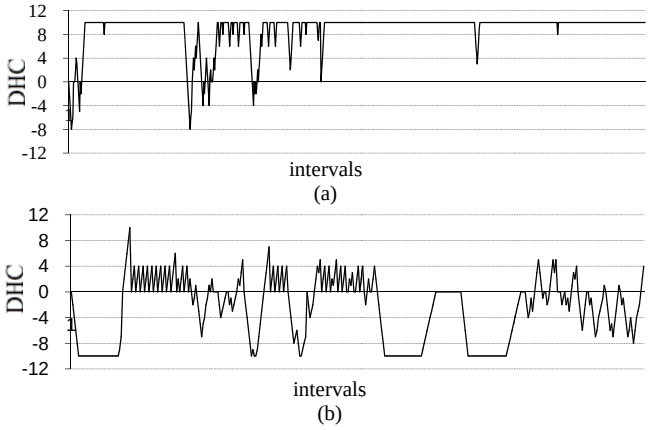

Figure 4. Demand counter values for a) a strong core and b) a weak core. The x-axis is the execution interval.

4. Methodology

We simulate a heterogeneous multicore processor using three types of cores with different performance levels, one EV6-like, two EV5-like and two EV4-like cores as reported in Table 1. Although there are many other possible configurations, our study shows that this configuration provides adequate execution resources to run the workloads chosen in this study.

The EV6-like core has the highest performance. The level one cache is private for each core and the instruction and data caches are separated. A large unified level two cache is shared between all cores. MESI protocol is used for cache coherency. All cores are simulated in 100 nm technology and run at 2.1 GHz.

Table 1. Core configurations

Core	EV6-like	EV5-like	EV4-like
Issue-width	6 (OOO)	4 (OOO)	2 (OOO)
IL1-cache	64 KB, 4 way	32 KB , 2way	16 KB , DM
DL1-cache	64 KB, 4 way	32 KB, 2 way	16 KB, DM
L2-cache	4 Mb , 8 way	(shared)	
B-predictor	Hybrid	Hybrid	static
Int Window size	104	80	56
FP window size	48	32	16

The simulations have been carried out using a modified version of the SESC simulator [7]. We use eight scientific/technical parallel workloads from Splash2 [8]. These workloads consist of four applications, i.e., barnes, water-spatial, ocean and fmm, and four computational kernels, i.e., radix, lu, cholesky and fft. We have used multi-program workloads which are composed of different sets of Splash2 benchmarks. Table 2 shows the benchmarks of each multi-program workload.

5. Experimental results

In this section we present the simulation results. To provide better understanding we also compare HARD to a static scheduler. However there are many possible static schedules possible for each application mix, we assign applications to cores based on application run time. For example, the most time consuming application is assigned to the strongest core. The application runs on the same core during the entire runtime.

Quantifying the performance of a computer system executing multiple programs is not a straightforward task as programs of the same mix interfere. In [2], Eyerman et al., introduced three metrics for quantifying the performance of a computer system executing multiple programs. Average *Normalized Turnaround Time* (*ANTT*) quantifies the average turnaround time slowdown due to multi application execution. *System Throughput* (*STP*) quantifies accumulated single-program performance under multi program execution. A system is fair if the coexecuting programs in multiprogram mode experience equal relative progress with respect to single-program mode.

5.1. Performance Oriented Configuration

In this section we report results assuming that performance is the main goal. To estimate performance of each mix we measure performance for each core and report average performance across all cores. We tune the scheduler using the parameters reported in Table 3. We run all mixed benchmarks using both HARD and the static scheduler. We measure and report (in Table 3) average performance and power achieved for all mixed benchmarks.

Table 2. Mixed benchmarks

	barnes	water-spatial	ocean	fmm	radix	lu	cholesky	fft
mix1		✓	✓		✓		✓	✓
mix2	✓	✓			✓	✓	✓	
mix3	✓	✓			✓		✓	✓
mix4		✓	✓		✓	✓	✓	
mix5	✓	✓	✓		✓		✓	
mix6	✓	✓	✓		✓	✓		
mix7		✓		✓	✓	✓	✓	
mix8	✓		✓		✓	✓	✓	

Table3. Thresholds for performance oriented scheduling.

Thr.	Core	LTT	LUT	HTT	HUT	Performance improvement	*Increased power*
Set-A	EV6-like	30% max	30%	50% max	70%	9%	*3%*
	EV5-like	30% max	30%	50% max	70%		
	EV4-like	30% max	30%	50% max	70%		
Set-B	EV6-like	25% max	20%	50% max	90%	10%	*2%*
	EV5-like	25% max	20%	50% max	90%		
	EV4-like	*25% max*	*20%*	*50% max*	*90%*		

In Table 3 max refers to the maximum possible throughput among all applications running on each core. For example, to measure max for the strongest core we run the eight benchmarks used in this study on the core. Among all benchmarks, water-spatial shows the highest throughput, deciding max.

Figures 5 and 6 show performance and power improvements achieved by HARD compared to the static scheduler, respectively. In the interest of space we only report for Set-B. For some benchmarks (mix4-6) both performance and power are improved by HARD. For other benchmarks, we witness an increase in power as the result of an aggressive usage of strong cores to achieve higher throughput. For instance, mix8 shows better performance compared to mix 4 as reported in Figure 5. Mix8, however, also shows higher power dissipation (Figure 6). This difference is the result of the benchmarks composing the mixes. In this case water-spatial benchmark (from mix8) is more performance demanding compared to barnes (from mix4).

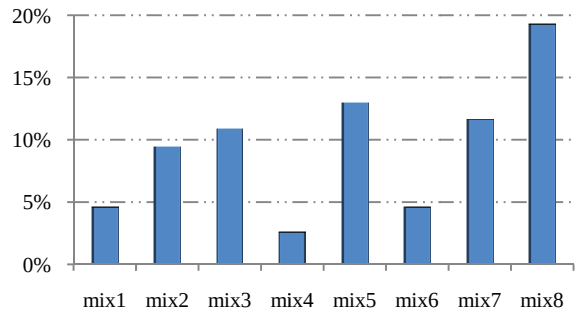

Figure 5. Performance improvement using HARD under the performance orineted configuration vs. static scheduling.

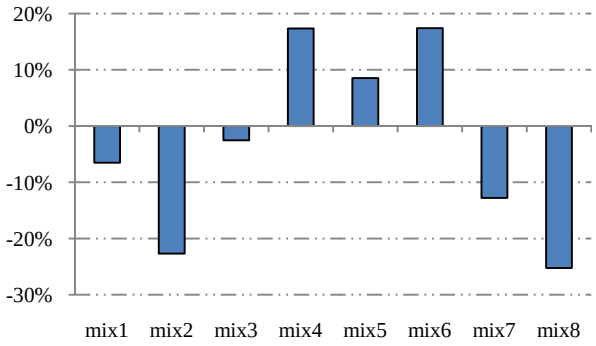

Figure 6. Power reduction achieved using HARD under the performance oriented configuration vs. static scheduling.

Figure 7 reports *ANTT*, *STP* and *Fairness*. Note that we use HARD for both single-application and multi-application executions. In single application execution, there is only one benchmark running on the processor. Every phase change of the benchmark will result in a switch to another core (one core is in use and all other cores are idle). On the contrary, in the multi-application mode, a request from an application does not necessarily lead to a switch as a switch also depends on the availability of target cores. Therefore, in this circumstance *ANTT* can be less than one. Note that an *ANTT* below one indicates that processor's turnaround time is improved comparing multi-program to single program execution. The same could be said about processor throughput when *STP* is greater than five (the number of applications in mixed benchmarks). Complete *Fairness* is achieved when *Fairness* is equal to one. As Figure 7 shows, HARD improves *ANTT* and *STP* for the mixes.

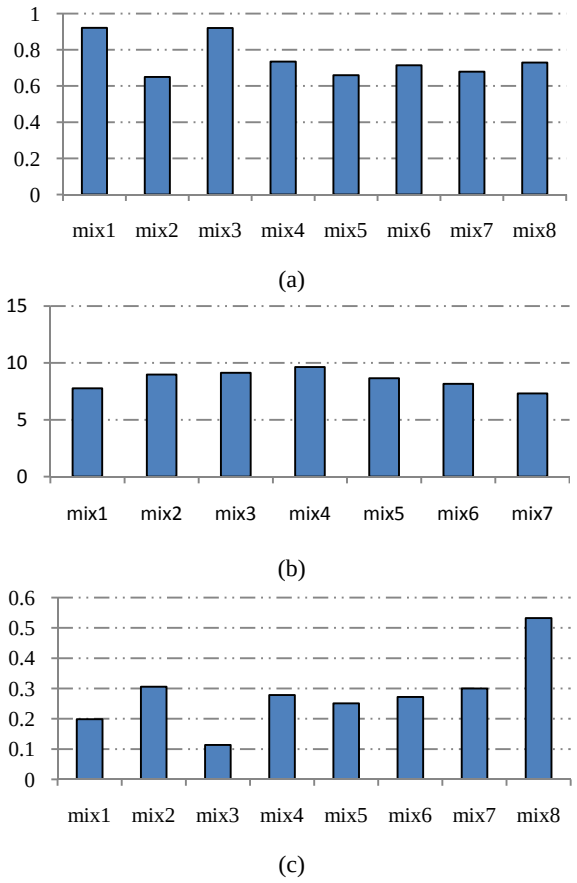

Figure 7. a) *ANTT*, b) *STP* and c) *Fairness* for HARD under the performance oriented configuration.

5.2. Power Oriented Scheduling

In this section we report results assuming that reducing power is the main goal. Again we measure and report average power across all cores. We use the parameters introduced in Table 4. Table 4 shows three sets of thresholds. As reported Set-C results in maximum power saving while Set-A leads to minimum performance loose. We run all mixed benchmarks using both HARD and the static scheduler. We measure and report (in Table 4) average performance and power achieved for all mixed benchmarks.

Figure 8 and 9 show the amount of performance loss and power saving, respectively, using Set-B from Table 4. As reported some mixes of benchmarks (e.g., mix7) come with higher performance loss but also show very high power reduction compared to others. On the other hand there are mixes (e.g., mix4) that show considerable power savings at the expense of modest performance loss.

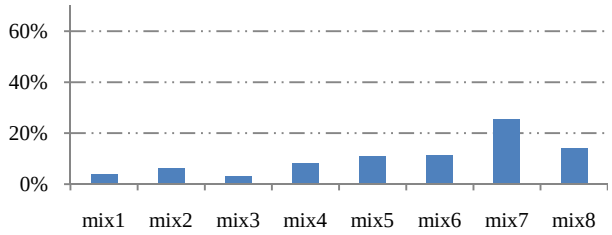

Figure 8. Performance loss using power oriented HARD scheduling vs. static scheduling.

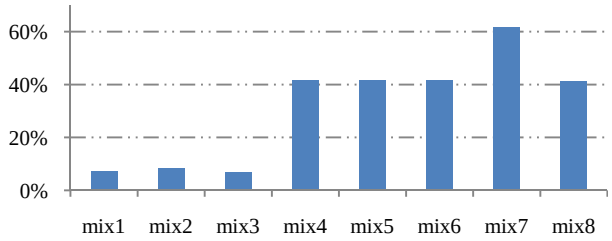

Figure 9. Power reduction achieved using HARD under the power oriented configuration vs. static scheduling.

Table4. Thresholds used in power oriented scheduling

Thr.	Core	LTT	LUT	HTT	HUT	Performance lost	*Power saving*
Set-A	EV6-like	50% max	35%	80% max	90%	9%	*30%*
	EV5-like	40% max	30%	70% max	70%		
	EV4-like	25% max	15%	60% max	60%		
Set-B	EV6-like	30% max	40%	90% max	90%	11%	*33%*
	EV5-like	25% max	25%	75% max	80%		
	EV4-like	20% max	30%	70% max	90%		
Set-C	EV6-like	40% max	30%	70% max	85%	21%	*46%*
	EV5-like	30% max	30%	80% max	80%		
	EV4-like	*25% max*	*25%*	*50% max*	*70%*		

Figure 10 reports *ANTT*, *STP* and *Fairness.*

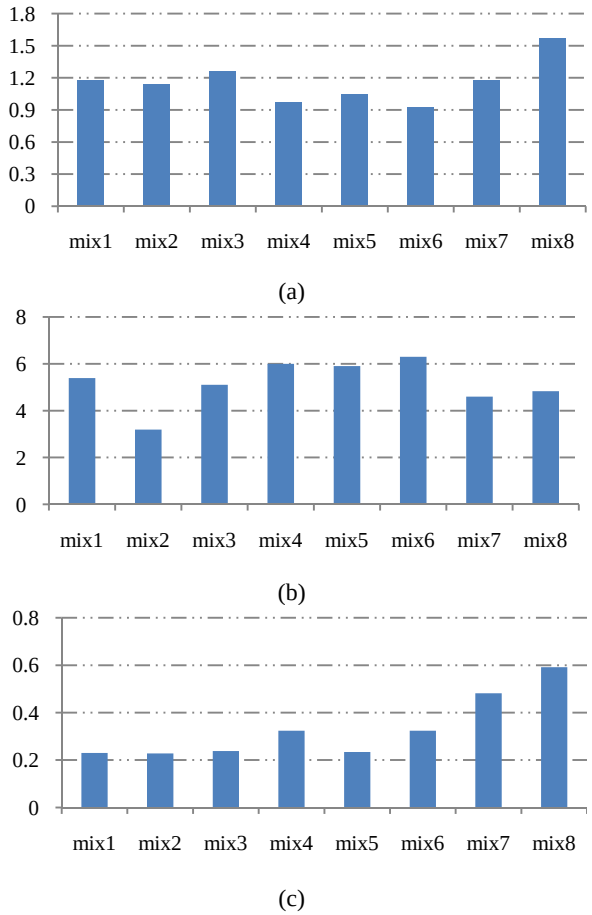

Figure 10. Average Normalized Turnaround Time, System Throughput and *Fairness* for HARD using power oriented configuration.

To provide better understanding we measured the number of UG/DG switches for mix2 and mix7 of Table2. The number of UG/DG switches of mix2 are 1/1, 1/0, 1/0, 2/2 and 2/1 for performance oriented scheduling and 0/0, 3/2, 0/0, 1/2 and 3/2 for power oriented scheduling for radix, lu, barnes, water-spatial and cholesky benchmarks, respectively. The number of UG/DG switches of mix7 are 1/1, 1/0, 1/1, 1/1 and 1/0 in performance oriented scheduling and 0/0, 5/4, 1/3, 7/7 and 4/2 in power oriented scheduling for radix, cholesky, water-spatial, fmm and lu benchmarks, respectively.

In Figure 11 we show the cores participating in the execution of barnes benchmark chosen from mix3. Regular and dotted lines show the cores picked to execute the application for both performance oriented and power oriented configurations respectively. For example, barnes starts its execution on the middle performance core and switches into the weakest core and ends its execution on the middle performance one under the power oriented configuration (represented by dotted line).

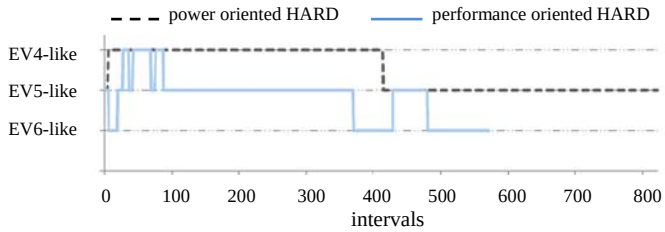

Figure 11. Cores that barnes benchmark meets during its execution with performance oriented (regular line) and power oriented (dotted line) HARD, extracted from mix3.

6. Related Works

In this section we briefly review previous works on phase change detection techniques and dynamic scheduling in multiprocessor, multithreaded and multicore platforms. Comparison between our proposed scheme and previous studies is part of our ongoing research.

In [9] authors improved the execution time and power of multicore processors by predicting the optimal number of Threads depending on the amount of data synchronization and the minimum number of threads required to saturate the off-chip bus.

In [10] Accelerated Critical Sections (ACS) is introduced which leverages the high-performance core(s) of an Asymmetric Chip Multiprocessor (ACMP) to accelerate the execution of critical sections. In ACS, selected critical sections are executed by a high-performance core, which can execute the critical section faster than the other, smaller cores. Consequently, ACS reduces serialization.

In [11] authors proposed scheduling algorithms based on the Hungarian Algorithm and artificial intelligence (AI) search techniques. Because of dynamic heterogeneity, hard errors and process variations, performance and power characteristics of the future large-scale multicore processors will differ among the cores in an unanticipated manner. These thread assignment policies effectively match the capabilities of each degraded core with the requirements of the applications.

In [12] devised phase co-scheduling policies for a dual-core CMP of dual-threaded SMT processors was introduced. They explored a number of approaches and find that the use of ready and in-flight instruction metrics

permits effective co-scheduling of compatible phases among the four contexts.

In [13] authors proposed a scheme for assigning applications to appropriate cores based on the information presented by the job as an architectural signature of the application.

In [14] authors made a case that thread schedulers for heterogeneous multicore systems should balance between three objectives: optimal performance, fair CPU sharing, and balanced core assignment. They argued that thread to core assignment may conflict with the enforcement of fair CPU sharing. They demonstrate the need for balanced core assignment. In [15] authors introduced a cache-fair algorithm which ensures that the application runs as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated. If the thread executes fewer instructions per cycle than it would under fair cache allocation, the scheduler increases that thread's CPU time slice.

In [16] Kumar monitored workload run-time behavior and detected significant behavior changes. The study also considered different trigger classes based on how IPC changes from one steady-phase to another.

7. Conclusion

In this work we presented HARD a dynamic scheduler for heterogeneous multi-core systems. HARD uses past thread assignments to find the best matching core for every core. HARD saves power by downgrading applications with low resource utilization to weaker cores. HARD improves performance by upgrading demanding application to stronger cores.

We study different program mixes and show that HARD can reduce up to 46% power while improving performance by 10% for the application mixes used in this study.

ACKNOWLEDGMENTS

This work was supported by the Iran's Institute for Research in Fundamental Sciences (IPM).

References

[1] W. Madison, A. Batson, Characteristics of program localities, In Communications of the ACM, vol. 19(5), May 1976, pp. 285-294.

[2] Eyerman, S.; Eeckhout, L. System-Level Performance Metrics for Multiprogram Workloads. IEEE Micro. IEEE Computer Society. Vol. 28 (3). 2008. pp. 42-53.

[3] P. Denning, On modeling the behavior of programs, In Proc. AFIPS Conf. 40 (SJCC), 1972, pp. 937-944.

[4] Batson, W. Madison, Measurements of major locality phases in symbolic reference strings, In Proc. 1976 International Symposium on Computer Performance and Modeling, Measurement and Evaluation, ACM SIGMETRICS and IFIP WG7.3, March 1976, pp. 75-84.

[5] T. Sherwood, E. Perelman, B. Calder, Basic block distribution analysis to find periodic behavior and simulation, In Proc. 2001 International Conference on Parallel Architectures and Compilation Techniques, Sep. 2001, pp. 3-14.

[6] Rakesh Kumar, Dean Tullsen, Parthasarathy Ranganathan, Norman Jouppi, and Keith Farkas. "Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance". In the 31st International Symposium on Computer Architecture, ISCA-31, June, 2004.

[7] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, P. Montesinos, SESC simulator, January 2005, http://sesc.sourceforge.net.

[8] S.C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, In Proc. 1995 International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, by the ACM, June 1995, pp. 24-36.

[9] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt, Feedback Driven Threading: Power-Efficient and High-Performance Execution of Multithreaded Workloads on CMPs, In the International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS) 2008.

[10] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi and Yale N. Patt, Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures, In the International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS) 2009.

[11] Jonathan A. Winter, David H. Albonesi: Scheduling algorithms for unpredictably heterogeneous CMP architectures. DSN 2008: 42-51

[12] Ali El-Moursy, R. Garg, David H. Albonesi, Sandhya Dwarkadas: Compatible phase co-scheduling on a CMP of multi-threaded processors. IPDPS 2006.

[13] Daniel Shelepov and Alexandra Fedorova, Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures, In Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA-35, Beijing, China, 2008.

[14] Alexandra Fedorova, David Vengerov and Daniel Doucette, Operating System Scheduling on Heterogeneous Core Systems, In Proceedings of the First Workshop on Operating System Support for Heterogeneous Multicore Architectures, at PACT 2007, Brasov, Romania.

[15] Alexandra Fedorova, Margo Seltzer and Michael D. Smith, Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler, In Proceedings of the Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), Brasov, Romania, September 2007.

[16] Rakesh Kumar, Holistic Design for Multi-core Architectures, PhD Thesis, university of California, San Die go. 2006.